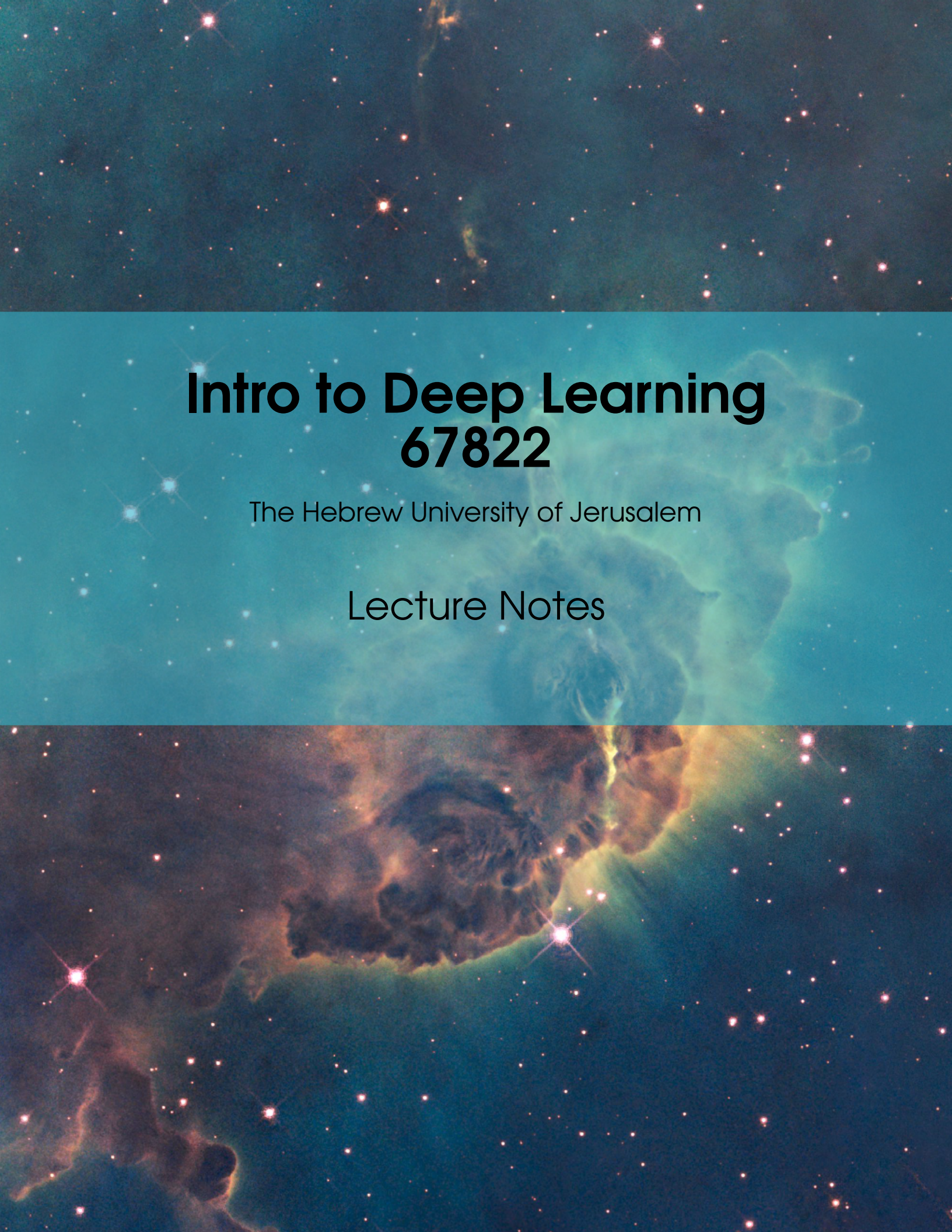


Intro to Deep Learning 67822

The Hebrew University of Jerusalem

Lecture Notes

WRITTEN BY HADAR SHARVIT

CONTACT ME AT - HADAR.SHARVIT1@MAIL.HUJI.AC.IL

Based on the lectures given by Raanan Fattal, and the recitations given by Matan Halfon.

Quick note before we start - the course does not intend to be formal, and nor does this summary. Having said that, I may elaborate on some topics I find important, and those cases will be marked with a (†) sign.

Last Update: December 3, 2021

Contents

1. Intro to Information Theory

Suppose we are given with data points $x \in X$, which are sampled from some unknown probability space P. One might ask - how would I find a somewhat similar probability space Q, that represents P well enough? in other words, what metric should I define to distinguish if Q resembles P? lucky for us, this has been studied long before neural networks walked the earth, and is related to the theory of information.

1.1 Entropy and Cross-Entropy

Definition 1.1.1 — Shannon's Entropy. for a discrete probability distribution p, we define the (Shannon) Entropy as

$$H(P) = -\sum_{x \in X} p(x) log_2(p(x)) = \mathbb{E}_{x \in X} log(1/p(x))$$

A "hand-wavy" definition to Entropy would be the amount of uncertainty a system has, that is - Entropy is an amount that represents how our data is unexpected when sampled.

■ **Example 1.1** as a simple example, consider three boxes with 4 balls in each. denoting $b_1 = \{R,R,R,R\}, b_2 = \{R,R,R,B\}$ and $b_3 = \{R,R,B,B\}$. in b_1, it is easy to guess the outcome of sampling one ball, and in b_3 - not so much. when it is "easy" to guess the outcome, the entropy is low and vice versa. Notice how this relates to the definition of uncertainty - in b_1, everything is certain as all balls have the same color R (therefore the entropy is low). In $b3$ on the other hand the entropy is high, as the uncertainty is high (maximal in fact, as there is 50% chance for either R or B). ■

going past the basic definition, the entropy also bounds from below the minimal number of bits that are needed to represent our data. for example, if our data is comprised of images, every image (sample) can only be compressed to a size that is exactly the entropy of our sample. What is the entropy of an image you ask? Well, we can define a probability function on some pixel to provide the chance to be equal to some grey-scale value. the summation of all the pixels would give the entropy of the image (but that's a topic for another time).

it is important to notice that as of now, we haven't answered the question that was presented in

the opening section. More specifically, the entropy is a function of a single probability space, and doesn't provide us with some way to compare two probability spaces as we wished for. For that we will define a new method.

Definition 1.1.2 — Cross Entropy (CE). for two discrete probability distributions p and q, we define the cross Entropy as

$$H(p,q) = -\sum_{x\in X} p(x)log_2(q(x)) = \mathbb{E}_{x\sim p(x))}log(1/q(x))$$

This definition measures the average number of bits needed to identify an event drawn from the set if a coding scheme used for the set is optimized for an estimated probability distribution q, rather than the true distribution p. some notes that are worth mentioning:

- we can refer to p as the target probability (as in the probabilities that are associated with the true labels), and q would be our approximation of p.
- the cross entropy does not obtain the value 0 when $p = q$, like we would have wanted. For that reason (and others) we define the `Kullback Leibler (KL)` divergence $D(p||q)$, that does achieve value 0 when the probabilities are the same:

$$D(p||q) = \sum_{x\in X} p(x)log(p(x)/q(x)) = \mathbb{E}_{x\sim p(x)}log(p(x)/q(x))$$

the cross entropy defines in many cases the loss function which we wish to minimize, and this fact is strongly related the the principle of `maximum likelihood` (denoted $\mathfrak{L}$, where we try to find a set of parameters w that are correlated to the highest probabilities

1.1.1 minimizing the CE = maximizing the Likelihood

this section is denoted (†).

as we clean the dust from our basic ML course, a bird may whisper to you that given parameters w, a sample matrix X and true labels y, the maximum likelihood principle wishes to find the most probable w's for such X, y. in a more formal way, we have

$$\mathfrak{L}(w|X,y) = Pr[y_1,...,y_m|X,w] \overbrace{=}^{x\sim iid} = \prod_i Pr[y_i|x_i,w]$$

as we wish to maximize the term, applying `log` wouldn't matter, and so does multiplying by some constant. so we denote $\mathfrak{l} = \frac{1}{|X|}log(\mathfrak{L})$, and from here:

$$\mathfrak{l}(w|X,y) = \frac{1}{m}\sum_{i=1}^{m} logPr(y_i|x_i,w) = \mathbb{E}_{x\sim p(x}logPr(y_i|x_i,w)$$

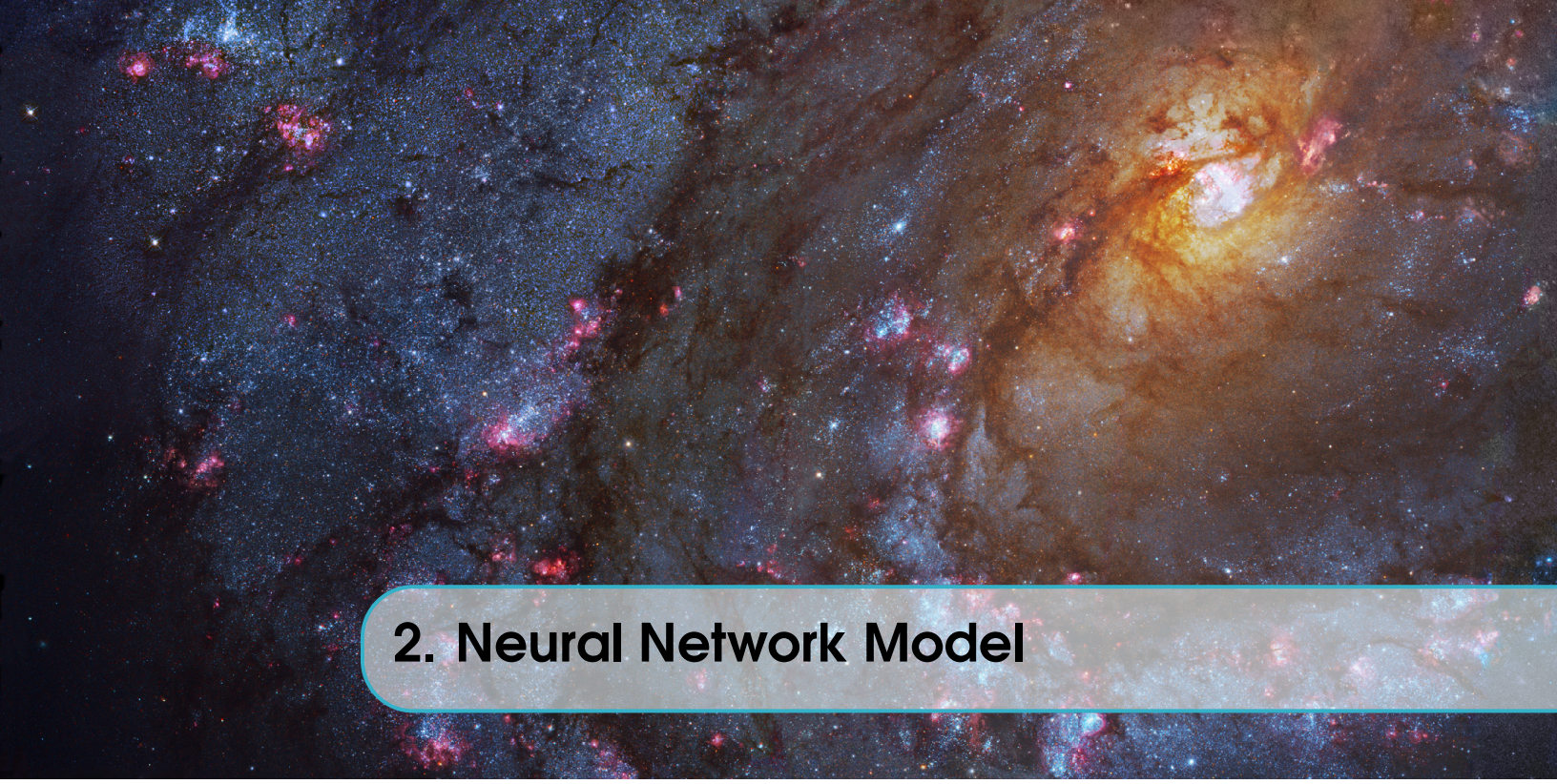
the latter may be a bit complex, but the meaning of it is that given some true label y_i, we ask how probable is the that we could represent it given our sampled $x \sim p(x)$, and provided the currently initialized parameters w. this formally mean that we can write the above as

$$\mathfrak{l}(w|X,y) = \mathbb{E}_{x\sim p(x)}log(q(x))$$

now our goal is to find some $\hat{w}$ such that $\hat{w} = argmax_w\mathfrak{l}(w|X,y)$, or in other words:

$$\hat{w} = argmin_w -\mathbb{E}_{x\sim p(x)}log(1/q(x)) = H(p,q)$$

which is what we wanted to show.



2. Neural Network Model

before we dive to the model of a neural network (aka a multi layer Perceptron), we will first remind ourselves what the Perceptron exactly is

2.1 Perceptron

this section is denoted (†)

the Perceptron is a supervised learning algorithm for binary classification, that finds a hyper-plane which splits a data-set S into two distinct sections/clusters. to formally describe to approach, let us first define some useful notations

Definition 2.1.1 — half-plane. we define a half-plane to be the group of points x as followed

$$\{x : \langle w,x\rangle = b, x \in \mathbb{R}^d\}$$

which is equivalent to

$$\{x : sign(\langle w,x\rangle - b) \geq 0), x \in \mathbb{R}^d\}$$

■ **Example 2.1** consider the weights $w = \{1,2,3\}$ and the bias $b = 0$. Those define the plane $\langle w,x\rangle = x_1 + 2x_2 + 3x_3 = 0$, or perhaps with relation to 3D space $x + 2y + 3z = 0$. This defines a plane that intersects with the point $(0,0,0)$. we can ask ourselves - given the points $a = (1,2,3)$ and $b = (-1,-2,-1)$, how are they positioned in relation to the plane defined by w and b. it is easy to verify that a is "above" the plane $(+1)$, and b is "below" it (-1), as figure 2.1 portrays ■

let $\mathbb{H}_{half} = \{h_{w,b}(x) = sign(x^T w + b) : w \in \mathbb{R}^n, b \in \mathbb{R}\}$ be an hypothesis class that represents all half-planes. we would like to find some $h \in \mathbb{H}_{half}$ that classifies every sample x from our data-set S to some value $\in \{+1,-1\}$ (note that this might not always be possible). The main approach uses the fact that when ever there's a "mismatch" between the signs of a true label y_i and $h(x_i)$ (for any i), which is the same as saying that $y_i\langle x_i, w\rangle \leq 0$, the weights are altered in a manner that decreases the chance for such mismatch to occur in later iterations. This can be formalized as followed: we initialize the weights to $w_0 = (0,0,...,0)$. Then, if $\exists i$ s.t $y_i\langle x, w_t\rangle \leq 0$ we update $w_{t+1} = w_t + y_i x_i$.

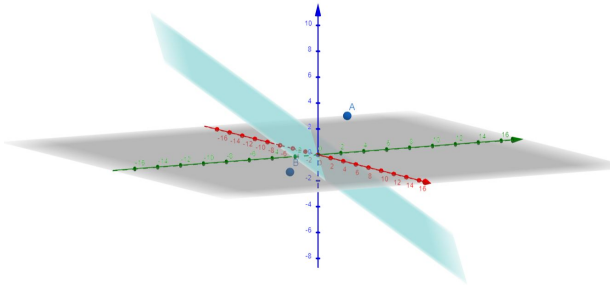


Figure 2.1: A hyper-plane in 3D

notice that $y_i\langle x_i, w_{t+1}\rangle = y_i\langle x_i, w_t + y_i x_i\rangle = y_i\langle x_i, w_t\rangle + y_i^2||x_i||$. that is, in every iteration we add an element $||x_i|| \geq 0$, which guarantees there exists some t for which the if condition doesn't hold, and in that case we return w_t.

2.2 Multi Layer Perceptron (MLP)

The term multi-layer Perceptron refers to a number of single Perceptrons that are organized to a layer. In addition, the MLP has more "freedom" to it, as it is equipped with a non-linearity function that enhances the expressibility of the model, and in general can also be used in regression problems (as opposed the the original Perceptron). In most books, the model of a single "neuron" is represented with the diagram in figure 2.2

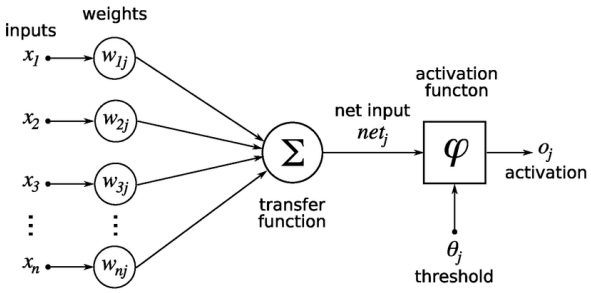


Figure 2.2: A single neuron

which indicates that given some input vector with n features $x = (x_1, x_2, ..., x_n)$, every value x_i is multiplied with some weight w_{ij}, that should in theory represent how valuable the $i'th$ neuron to the learning process, such that the sum of all of those is the input to some activation function ϕ, to which we also add a threshold θ_j (that somewhat resembles to the activation notion of an actual neuron). This long sentence could be summarized to a single formula to the activation o_j of the $j'th$ neuron (which will resemble the notation for the Perceptron of the previous section:

$$o_j = \phi(\textstyle\sum_{i=1}^{n} w_{ij}x_i + \theta_j) = \phi(\langle w_j, x\rangle + \theta_j)$$

where w_j in the single neuron model is a vector that represents the weight associated our neuron (j) and all of the other neurons in the next layer.

the generalization of the above to many neurons is straight forward, as the $(i+1)'th$ layer x_{i+1} is represented using the previous layer x_i and a now weights matrix W, such that $w_{ij} = [W]_{ij}$ indicates the weight associated with the edge (i, j). formally we can write $x_{i+1} = \phi(W_i x_i + \theta_i)$ (note that though ϕ here is the same for every neuron, this is not obligatory). Before moving on, notice that

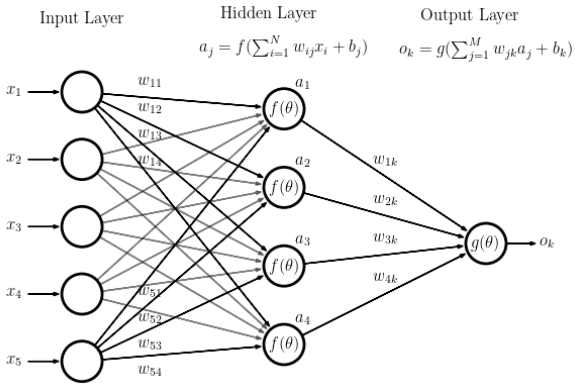


Figure 2.3: A Neural Network diagram.

as much as the NN model diagrams are very cool, all they represent eventually is a series of affine transformations (with the addition of the non-linearity). That is, the output of a MLP is simply a set of affine transformations + non-linearities, and that doesn't require any diagram.

2.3 Activation Function

an activation function ϕ is a function (mostly non linear one) that is given the activation of a neuron as an input, and outputs some number. The fact that those function are mostly non-linear is am important one, as it is what distinguishes the neural network model from a basic model, only capable of predicting linear phenomenons from a much more capable one. In other words, the non-linear activation function allows the model to predict/classify various range of cases, that are not necessarily linear in nature (this will be formulated later on). Lastly, in relation to the previous sentence, notice that when using linear activation functions, the out come is a linear component, and when summed over all the layers network, we are left with an output that can be represented as $x_{out} = \mathfrak{w}x_{in} + \mathfrak{b}$, where $\mathfrak{w}$ and $\mathfrak{b}$ are some combinations of the original $\{W_i, b_i\}_{i=1}^{n}$. This simply indicates that when using a linear model we lose the significance of the layer formation (as well as strict our selves only to linear cases, as stated before).

2.3.1 Types

- `sigmoid`: also called `logistic`. defined as

$$\phi(x) = \frac{1}{1+e^{-x}}$$

  this one maps all values x to some normalized value $\in (0,1)$
- `relu`: which stands for rectified linear unit.

$$\phi(x) = max(0,x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

obviously, many more options exist, but those are the basic (and most known for) ones. Furthermore, every function may be better or worse depending on the task in hand. for example, `sigmoid` will not describe magnitudes, but is valuable when the output should resemble distribution (as it is normalizing the values). `relu` is useful in many cases, but may cause problems if we are dealing with negative activations, and in that case the function outputs 0, which may cause problem in `SGD` process (small or 0 grad may affect convergence)

2.4 Loss Function

The following step to building a NN model would be to come up with a meaningful loss function, that will somehow provide us with a number that represents how "good" our prediction is. This loss is sometimes called the `"Empicical Risk"`, and is defined as followed:

Definition 2.4.1 — Empirical Loss. given algorithm h (in our case, the algorithm is the network), and m samples $x_i \in \mathbb{R}^d$ the `ER` is

$$ER(h) = \frac{1}{m}\sum_{i=1}^{m} l(h(x_i), y_i)$$

where $l : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

remember that our "goal" is to find $\hat{h}$ such that $\hat{h} = argmin_{h \in H} ER(h)$, and H is the hypothesis class. this is the `empirical risk minimization (ERM)` rule. The method to achieve such minima will be discussed later on.

2.4.1 Regression Loss

when a specific number is to be predicted (for example - predicting housing prices, weight percentage, etc..), a good rule of thumb would be that a small/big deviation from the "true label" would yield small/big loss respectively. One option to do that is using l_2 norm loss. Of course, other forms of functions l could work, depending on the problem. In formal notation, if we denote out hypothesis as $h = N_\theta$, where θ represents the network parameters, and use l_2 norm we have

$$ER(N_\theta) = \frac{1}{m}\sum_{i=1}^{m} ||N_\theta(x_i) - y_i||_2^2$$

2.4.2 Classification Loss

When the task is hand is to predict finite number of classes, a naive approach could use a "misclassification" loss (sometimes called "l_0" norm, though isn't a norm per say, or "$0-1$" loss).

Definition 2.4.2 — Misclassification (0,1) Loss. given algorithm h, and m samples $x_i \in \mathbb{R}^d$ the `0,1loss` is

$$L_{0,1}(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(h(x_i) = y_i)$$

where $\mathbb{1}$ is the indicator function

the misclassification loss may be a bad idea in cases where it is important to distinguish `Type1` or `Type2` errors. Another option is to use `Cross Entropy`, which we have already seen before. For this to have any meaning though, we must first "cast" the network output to some bounded values, that would represent some distribution p. Two common metrics are (for a network output vector $\hat{y}_i$):

- `logistic`: $\hat{y}_i \mapsto 1/(1+e^{-\hat{y}_i})$ suited for binary classification

- `softmax`: $\hat{y}_i \mapsto e^{\hat{y}_i}/(\sum_i e^{-\hat{y}_i})$. mostly suited for multi class classification

after this was done, the cross entropy would give an estimation to the network's "capabilities", by comparing the true distribution of the data with the network's output distribution:

Definition 2.4.3 — Cross Entropy in Multi Class Classification. $H(p,q) = -\sum_{c=1}^{C}\sum_{i=1}^{m}\hat{y}_i log(h(x_i))$ where C is the number of classes.

specifically for binary classification, we have

$$H(p,q) = -\sum_{i=1}^{m}[\hat{y}_i log(h(x_i)) + (1-\hat{y}_i)log((1-h(x_i)))]$$

where $h(x_i)$ are the logistic values of the output labels.

2.5 Network Training

We have structured a neural network model N_θ, and chose a meaningful loss function L. Now its time to train the model, that is, find all the parameters θ that would minimize L. In other words, we wish to calculate $min_\theta\ L$, and a key component to this is the fact that we can find the (local) minimum iteratively, by taking "steps" in the direction of the maximum descent, aka the minus gradient direction $-\nabla L$

2.5.1 Gradient Descent Optimization

We would like to "move" in a direction $\Delta x \in \mathbb{R}^n$ such that $f(x_t + \Delta x) < f(x_t)$ (where t denotes the current iteration and f is the loss). usually we pick $\Delta x = -\eta \nabla f|_{(x_t)}$, where η is a step size (predefined hyper parameter). This choice is not arbitrary - if f is differentiable around x_t (if its not we use the sub-differential, though we will not linger on this) we can look at its Taylor approximation

$$f(x_t + \Delta x) = f(x_t) + \Delta x^T \nabla f|_{(x_t)} + O(||\Delta x||^2)$$

as we wish to maximize the difference in every step, our problem is

$$\max_{||\Delta x||=\varepsilon} f(x_t + \Delta x) - f(x_t) \approx \max_{||\Delta x||=\varepsilon} \Delta x^T \nabla f|_{(x_t)}$$

(for some small ε). using Lagrange multipliers for $h = \Delta x^T \nabla f|_{(x_t)}$ and $g = ||\Delta x|| - \varepsilon = (\Delta x)^T \Delta x - \varepsilon$:

$$\begin{aligned}\nabla_{(\Delta x)^T} h = \lambda \nabla_{(\Delta x)^T} g \rightarrow \nabla_{(\Delta x)^T}(\Delta x^T \nabla f|_{(x_t)}) &= \lambda \nabla_{(\Delta x)^T}((\Delta x)^T \Delta x - \varepsilon) \\ \rightarrow \nabla f|_{(x_t)} &= \lambda \Delta x\end{aligned}$$

What this really tells us, is that if we wish to maximize the objective, we should pick Δx that is in the direction of the gradient ∇f (which in general correlates with that we know about the gradient - direction of steepest ascend). the above can be summarized to an iteration loop which is sometimes referred to as `gradient descend process`:

Definition 2.5.1 — Gradient Descend Process. given a neural network model N_θ with parameters θ (weights,biases,etc..), step size η, true labels y, samples X and a loss function L, a gradient descend process over the parameters θ is

$$\theta_{i+1} = \theta_i - \eta \nabla_\theta L(N_{\theta_i}, X, y)$$

some important things to keep in mind

- the convergence isn't guaranteed to be towards a global maximum.
- the step size should be small enough to justify linear approximation, though not too small so we wont converge

2.5.2 Mini Batching and SGD

as our data becomes larger, it may be less feasible to calculate the gradient of the loss function with respect to every sample in every iteration. For that reason we may choose a subset of samples randomly, and find the gradients with respect to those samples. Notice that this does not come without risks, as our data may not be consistent, which may cause problems when trying to converge. Nevertheless, if we are sufficiently confident that every random sample can represent out data properly, the calculated gradient is this approach gives an approximation to the actual gradient, which in theory should bring us closer to a (local) minima.

formally speaking, if $B \subset X$ is a mini batch of samples that is randomly generated from our data X, the gradient of the loss function over all X is approximated by the gradient over all B:

$$L = \frac{1}{|X|}\sum_{i\in X} l(N_\theta(x_i), y_i) \Rightarrow \nabla L \approx \frac{1}{|B|}\sum_{i\in B}\nabla l(N_\theta(x_i), y_i)$$

from the section above, we can now formulate an algorithm

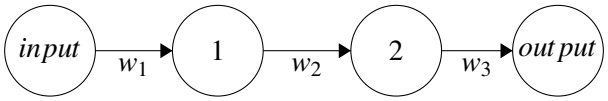
Algorithm 1 Stochastic Gradient Descent

Require: learning rate η, number of epochs T, batch size m, NN model N_θ
  $\theta \leftarrow (0,0,...,0)$
  for $t = 1,2,...,T$ **do**
    sample a mini-batch: $B = \{\mathbf{x_i}, y_i\}_{i=1}^{m}$
    $g \leftarrow \frac{1}{m}\sum_{i\in B}\nabla l(N_\theta(\mathbf{x_i}), y_i)$
    $\theta \leftarrow \theta - \eta g$
  end for

2.5.3 Back-propagation

this section is (mostly) denoted (†)

It turns out that the calculation of the gradient is in itself a quite daunting task. In this section we will see how to calculate it efficiently using what is known as the back propagation algorithm. lets start with a basic neural network - input node, two hidden nodes and one output node



in this network the loss is $L = L(w_1, b_1, w_2, b_2, w_3, b_3)$, and we wish to find what are the weights and biases that provide the minimal L. In the section below, we will seclude ourselves to l_2 loss (though this is generally true for every loss), the networks error over the $0'th$ sample (say, the first image out of $20,000$) is

$$L_0 = (a^{(L)} - y)^2$$

where $a^{(L)}$ is the activation of the $L'th$ layer (the last one), and y is our true labels vector. more specifically, the activation is (as stated at the beginning of the chapter) is

$$a^{(L)} = \sigma(w^{(L)}a^{(L-1)} + b^{(L)}) := \sigma(z^{(L)})$$

our goal is to understand how the loss L changes with respect to $w^{(L)}$, which can be written as its partial derivative

$$\frac{\partial L_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}}$$

we know the exact forms of L_0, $a^{(L)}$ and $z^{(L)}$, therefore

$$\frac{\partial L_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$
$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$
$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

which means

$$\frac{\partial L_0}{\partial w^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

note that this is only for the $0'th$ sample, and for the entire data-set we take the average:

$$\frac{\partial L}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial L_k}{\partial w^{(L)}}$$

now we repeat the process for the biases b:

$$\frac{\partial L_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}} = 1 \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

and when averaging:

$$\frac{\partial L}{\partial b^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial L_k}{\partial b^{(L)}}$$

Great! but those were just two elements in the vector ∇L we wish to calculate, as we remember that

$$\nabla L = \left(\frac{\partial L}{\partial w^{(0)}} \quad \frac{\partial L}{\partial b^{(0)}} \quad \cdots \frac{\partial L}{\partial w^{(L)}} \quad \frac{\partial L}{\partial b^{(L)}} \right)^T$$

so now we have to look at every layer $l \leq L$. we start with $l = L - 1$:

$$\frac{\partial L_0}{\partial w^{(L-1)}} = \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial L_0}{\partial a^{(L-1)}}$$

the first two elements are calculated by hand:

$$\frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} = a^{(L-2)}$$
$$\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = \sigma'(z^{(L-1)})$$

for the last element though, notice that When using chain rule again we get

$$\frac{\partial L_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}}$$

aside from $\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)}$, we already have the two right most elements, as they were calculated in the previous part (where $l = L$), so we could write

$$\frac{\partial L_0}{\partial w^{(L-1)}} = a^{(L-2)} \sigma'(z^{(L-1)}) w^{(L)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

in other words, to find $\frac{\partial L_0}{\partial w^{(L-1)}}$ we needed $\frac{\partial L_0}{\partial a^{(L-1)}}$, but the latter is already known from previous iterations. This would give us a recursive definition for $\frac{\partial L_0}{\partial w^{(l)}}$. for any $l \leq L$, but before that - lets generalize for a multi layered network:

we will add a sub-script that indicates which neuron are we referring to in the specific layer. for example: $a_k^{(l)}$ is the activation of the $k'th$ neuron of the $l'th$ layer. Now, L_0 is simply the sum over all neurons. suppose the $L'th$ layer has n_L neurons, so

$$L_0 = \sum_{j=0}^{n_L} (a_j^{(L)} - y_j)^2$$

lets denote the weight between $node_k^{L-1}$ and $node_j^L$ as w_{jk}^L. the activation would now be

$$a_j^{(L)} = \sigma(w_{j,0}^{(L)} a_0^{(L-1)} + ... + w_{j,n_L-1} a_{n_L-1}^{(L-1)} + b_J^{(L)}) = \sigma(\mathbf{w}_j^{(L)} \cdot \mathbf{a}^{(L-1)}) := \sigma(z_j^{(L)})$$

or simply

$$z_j^{(L)} = \sum_{k=0}^{n_{L-1}-1} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

we now have everything we need to find the derivatives

$$\frac{\partial L_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}} = a_k^{(L-1)} \sigma'(z_j^{(L)}) 2(a_j^{(L)} - y_j)$$

where as usual, the entire loss is an average

$$\frac{\partial L}{\partial w_{jk}^{(L)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial w_{jk}^{(L)}}$$

repeating the process for the bias:

$$\frac{\partial L_0}{\partial b_j^{(L)}} = \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}} = 1 \sigma'(z_j^{(L)}) 2(a_j^{(L)} - y_j)$$

and the average

$$\frac{\partial L}{\partial b_j^{(L)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial b_j^{(L)}}$$

following the thread of the single layer network, we would also wish to find the derivatives according to the activations. This is where the multi layer network differs from the single layer one. The main difference is that while for the single layer model there was only one activation, in the multi layer model every node has its own activation, and they all should be summed together (as the loss L is a function of them all). this can be formally written as

$$\frac{\partial L_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}}$$

This is pretty much all we need - so we can write it all together:

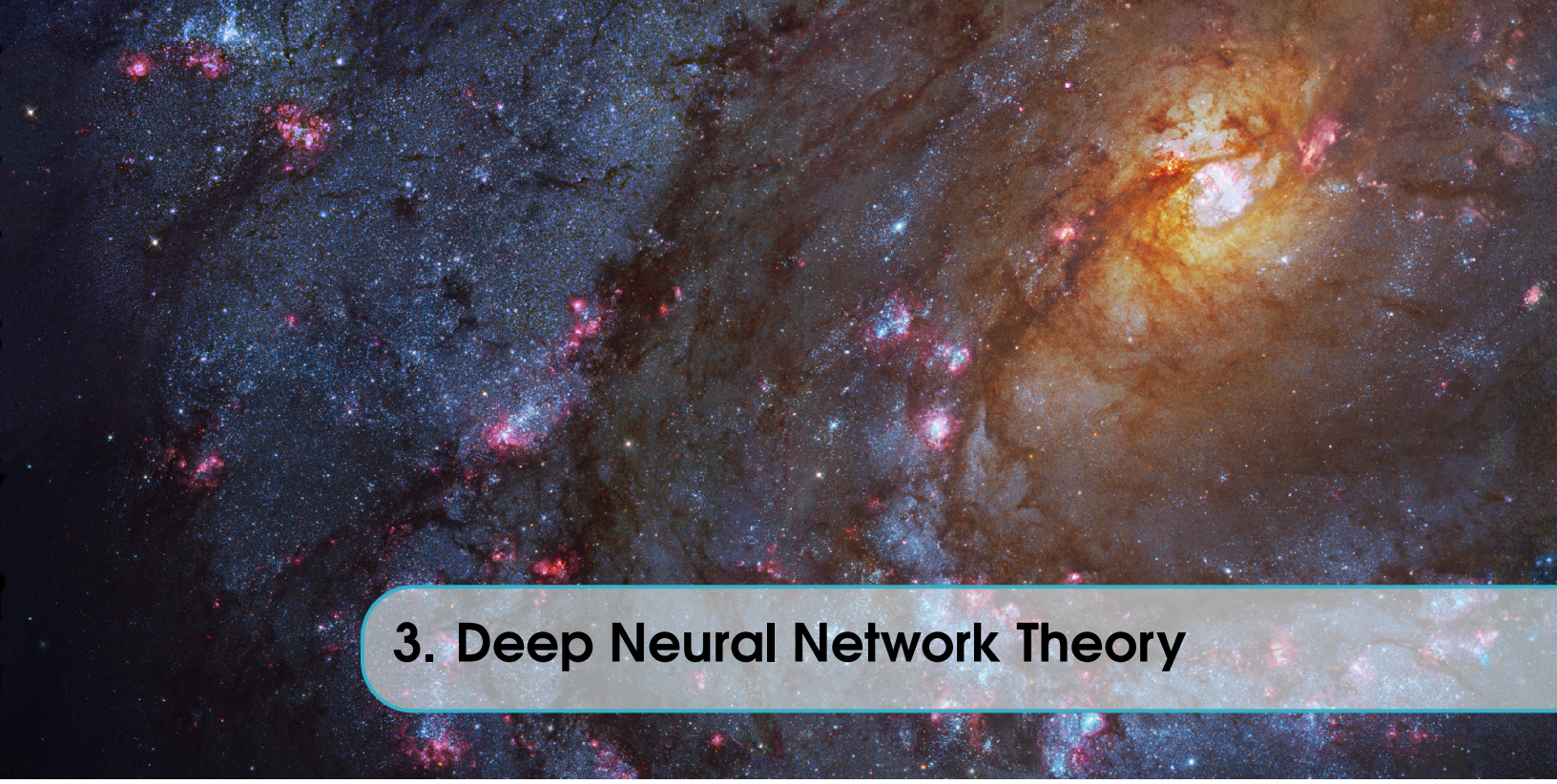
$$\frac{\partial L_0}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \sigma'(z_j^{(l)}) \frac{\partial L_0}{\partial a_j^{(l)}}$$
$$\frac{\partial L_0}{\partial b_j^{(l)}} = \sigma'(z_j^{(l)}) \frac{\partial L_0}{\partial a_j^{(l)}}$$

where

$$\frac{\partial L_0}{\partial a_j^{(l)}} = \begin{cases} \sum_{j=0}^{n_{l+1}-1} w_{jk}^{l+1} \sigma'(z_j^{l+1}) \frac{\partial L_0}{\partial a_j^{(l+1)}}, & l < L \\ 2(a_j^{(l)} - y_j), & l = L \end{cases}$$

finally, the elements of the vector ∇L are

$$\frac{\partial L}{\partial w_{jk}^{(L)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial w_{jk}^{(L)}}$$
$$\frac{\partial L}{\partial b_j^{(L)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial b_j^{(L)}}$$



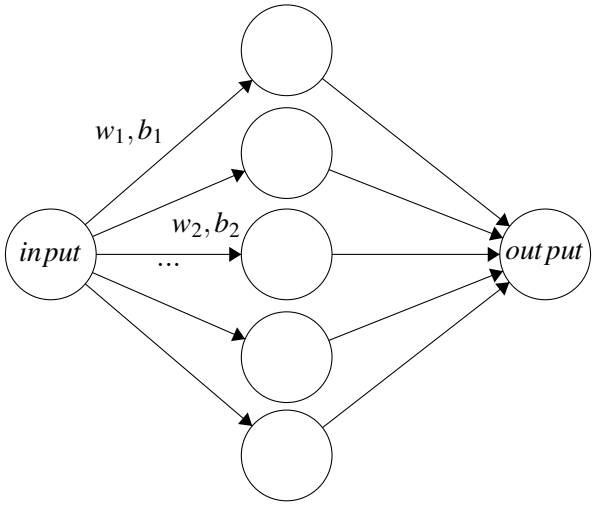
3. Deep Neural Network Theory

3.1 Universal Approximation Theorem

let us deal with the question of how "expressive" a neural network can get. More specifically, a network with only one hidden layer.

Theorem 3.1.1 — Cybenko 89. for a continuous monotone function σ with $\sigma(-\infty) = 0$ and $\sigma(\infty) = 1$, the family of function $f(x) = \sum_i \alpha_i \sigma(w_i x + b_i)$ is dense in $C([0,1])$ w.t.y the supremum norm $d(f,g) = sup|f(x) - g(x)|$

In a less formal form, Cybenko's theorem states that we can approximate any function g that is continuous and defined on $[0,1]$ (`sigmoid` for example) with arbitrarily small precision using a function f that represents a single hidden layer neural network. the network itself can be represented as below



notice that both input and output are scalars, and it that matter the network is "as simple as it gets".

A proof is not provided here, but intuitively speaking - the model can represent any stretch w_i, shift b_i and weight α_i to the input, and this is what gives the model its power or expressively. with respect to Cybenko's theorem, few notes are worth taking:

- the work of Cybenko was extended by Hornik 2 years later, and in his theorem the statement was provided for every bounded σ, which is a relaxation to the claim that σ must be monotone.
- the above does not contradict the fact that one can also use $\sigma \equiv relu$ function, even though `relu` is not bounded. this is because the following composition:

$$f(x) = relu(Wx+b_1) - relu(Wx+b_2)$$

is bounded, and for a specific choice of b_1, b_2, the function f is sigmoidal ("S" shaped, and bonded)

3.2 Shallow compared to Deep NN

A legitimate question that may arise from Cybenko's Theorem is - if a shallow layer can represent every function with small error, why should we go the extra mile by defining a deep network (with 2 or more hidden layers)? To answer this question, let us begin with a short example

■ **Example 3.1** let us define the `t-saw-tooth` function as a piece-wise affine function with t pieces. Also, we define the `hat` function as $hat(x) = relu(2relu(x) - 4relu(x-0.5))$. note that `hat` is a `4-saw-tooth` function by definition. We can use `hat` to concatenate `saw-tooth` functions - note that $t(x) = hat(x) + hat(x-1)$ comprises of two `hats`, and is a `6-saw-tooth` function, and the composition $t(t(x))$ is a `10-saw-tooth`, $t(t(t(x)))$ is a `18-saw-tooth`, and in general, a composition of T functions is a $(2+2^{T+1})$-`saw-tooth`, namely, the composition is comprised of 2^T hats.How can we represent this composition using a shallow network? remember that every neuron in the (single) hidden layer "represents" a single `relu`, therefore two neurons can represent a single `hat`. As there are exponentially many hats, a shallow network will have to use exponentially many neurons (2^{T+1} to be exact). On the other hand, a deep neural network would only need $\Theta(T)$ (if, for example, we use T layers and 2 nodes per layer). This is the case because the deep network $i'th$ layer receives as input the values of the $(i-1)'th$ layer, and for that matter is represents the composition of $t(x)$ simply as a product of its architectural design. ■

the example above demonstrates an exponential gap (in terms of size) between the shallow and deep NN architectures, and can be summarized to the following lemme:

Lemma 3.2.1 if f is `a-saw-tooth` and g is `b-saw-tooth`, $f+g$ is at most `(a+b)-saw-tooth` and $f(g)$ is at most `ab-saw-tooth`

"proof". as for $f+g$, the "worst-case-scenario" is when the edges of f and g doesn't intersect, and in that case $f+g$ will inherit all the discontinuities that f and g had. For $f(g)$, the image of each piece of g can contain the entire range in which f has all its pieces, which means that for every one of the b affine section of g we provide an input (to f) that generates a affine sections, to a total sum of $a \cdot b$. ■

the above statements could also be asked in reverse order - we have claimed that a deep layer with T layers and $O(T)$ neurons can be represented using a shallow layer with 2^T neurons, and now we claim that a shallow network with n neurons could be represented with a deep network, using $\Theta(n)$ neurons. As an example, we go back to our shallow network model $f(x) = \sum_i \alpha_i \sigma(w_i x + b_i)$ under the assumption $\alpha_i > 0$ and construct a deep network with n layers and $O(1)$ neurons per layer

as followed: the network is built from three main "branches" h_1, h_2 and h_3. h_1 will be used to "carry" the input x as it was initially provided, h_2 is for the actual calculation of the affine transformation on the input and h_3 sums the values of h_2 note that the first value of h_3 is $\sigma(x+L)$, where L is a shift

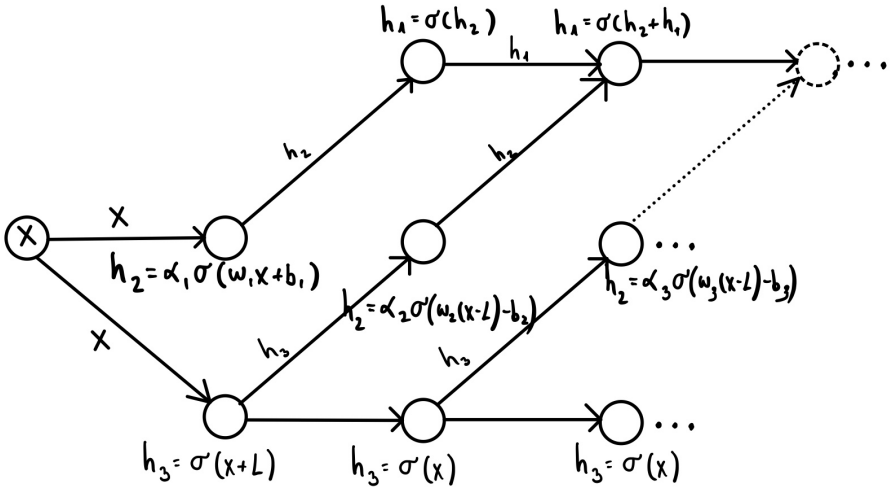


Figure 3.1: Shallow NN to Deep NN conversion.

we make to ensure positive values. This is needed so we wont lose any negative value edges under the `relu` function. In the following parts of h_2 we shift back to the original location $x-L$. Also note that as $\alpha_i > 0$, the values of h_3 are always positive, that is $\sigma(h_1+h_2) = h_1+h_2$. the network provided in the figure has $\Theta(n)$ layers with 3 neurons in each, which is what we intended to show ($O(n)$ neurons). All in all, we can summarize the above with the following diagrams

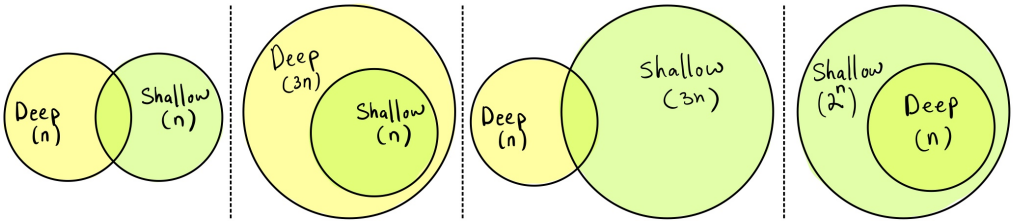
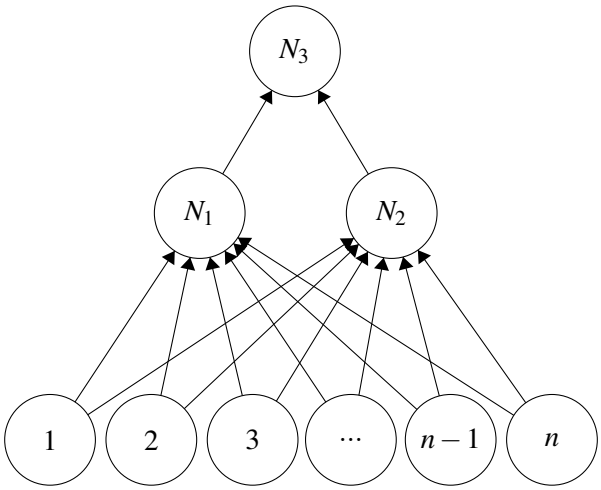


Figure 3.2: Venn Diagram of NN models with $(\cdot)$ neurons

3.3 Training a NN is NP-Complete

In this section we will prove that the training process of a NN is considered a NP-complete problem. The model we will use consists of Three neurons and is fully connected, where the input is of length n.



Moreover, the neurons N_1, N_2 and N_3 performs `Heaviside` activation on its input, as followed - given input $x = (x_1, ..., x_n)$, weights $(a_1^{(1)}, a_2^{(1)}, ..., a_n^{(1)}, a_1^{(2)}, a_2^{(2)}, ..., a_n^{(2)}, a_1^{(3)}, a_2^{(3)})$ and biases (b_1, b_2, b_3)

$$f_{i=1,2}(x) = \begin{cases} 1, & \sum_{j=1}^{n} a_j^{(i)} x_i + b_i > 0 \\ -1, & \text{else} \end{cases}$$

$$f_{i=3}(x) = \begin{cases} 1, & a_1^{(3)} N_1 + a_2^{(3)} N_2 + b_3 > 0 \\ -1, & \text{else} \end{cases}$$

where N_1, N_2 in $f_{i=3}$ indicates the outputs of neurons N_1, N_2, which are provided as input to N_3.

Corollary 3.3.1 — The Trainability Problem. given a set of $O(n)$ samples $\in \{0,1\}^n$, finding the Networks' parameters and threshold functions for which it produces outputs consistent with the training set is NP-complete

Proof. soon to be ■

4. Time/Translation Invariance

4.1 Notion of Time-Space

In this section we will discuss systems that are Invariant - either in time or in space translations. Such prior knowledge on our data will allow us to define more suited NN models that could potentially provide precise solutions using less parameters, and with great efficiency.

4.1.1 Translation/Time Invariant Systems

Systems that does not depend on time or spatial coordinated are called Translation/Time Invariant. This formally means that a shift in the input leads to a shift in the output, which for some operator L translates to:

$$L[f(x+t)](y) = L[f(x)](y+t)$$

where f is our input (image/audio for example). for example - our face detector algorithm should operate the same for all faces that are in different offsets in the image.

from here on we will refer to $f(x)$ as discrete samples of our signal, that is

$$f(x) = \sum_s \delta(x-s) f(s)$$

where $f(s)$ are scalars and δ is `Dirac delta function`

4.1.2 Linear Translation/Time Invariant Systems (LTI)

Let us also consider the linear and Time/Translation Invariant (TI) case, in which L performing on the sum of samples is as if we sum the operation of L on every sample by itself. This setup provides a convenient conclusion:

$$L[f(x)](y) = L[\sum_s \delta(x-s) f(s)] \overset{linearity}{=}$$
$$\sum_s f(s) L[\delta(x-s)](y) \overset{TI}{=}$$

$$\sum_s f(s)L[\delta(x)](y-s) \overbrace{=}^{g(x):=L[\delta(x)]} \sum_s f(s)g(y-s) = f*g$$

In other words, if L represents a `LTI` system, it can be written as convolution of the input f with some filter/kernel $g = L[\delta]$ (that can be chosen to our likelihood).

It is also worthy mentioning that every linear transformation can be expressed by a matrix. In our case

$$(f(x)*g(x))(y) = \sum_s f(s)g(y-s) = \langle f(x), g(y-x)\rangle = G\hat{f}$$

Where we restricted ourselves to finite space (sum is from $s=0$ to $n-1$) and cyclic convolution, where n signifies the number of elements in the signal, and there are infinitely many cycles of repeating n such elements. This is for

$$G = \begin{pmatrix} g(0) & g(1) & g(2) & ... & g(n-1) \\ g(n-1) & g(0) & g(1) & ... & g(n-2) \\ g(n-2) & g(n-1) & ... & ... & g(n-3) \\ \vdots & \vdots & ... & ... & \vdots \\ g(1) & g(2) & ... & g(n-1) & g(0) \end{pmatrix} \text{ and } \hat{f} = \begin{pmatrix} f(1) \\ f(2) \\ \vdots \\ f(n) \end{pmatrix}$$

notice that every row in G is a cyclic permutation of the previous row

■ **Example 4.1 — using G and $\hat{f}$.** denote $h = f*g$ and let f,g be defined in the coordinate range $[0,5]$ as followed: $f = (0,0,0,1,0,0)$ and $g = (0,0,0,1,-1,0)$, for specific $y = 6$

$$h(6) = \sum_{s=0}^{5} f(s)g(6-s) = f(0)g(6) + ... + f(3)g(3) + ... + f(5)g(1) = f(3)g(3) = -1$$

for every value of h, we can use G and $\hat{f}$:

$$h = \begin{pmatrix} 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

■

■ **Example 4.2 — derivative.** consider the filter $g = (-1,1)$, and some general $f(x)$ notice that $f*g = f(x+1) - f(x)$.

how does this relates to derivatives? from definition

$$\frac{\partial f(x)}{\partial x} = \lim_{\varepsilon\to\infty} \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$$

this can be approximated when f is discrete (for example, if f is some image, we can consider a single pixel as our minimum length, which has size 1). That is

$$\frac{\partial f(x)}{\partial x} \approx f(x+1) - f(x)$$

■

before we move on, let us introduce the `2D` convolution, as it is relevant for many convolutional NN architectures

Definition 4.1.1 — discrete 2D convolution. given 2D functions $f(x,y)$ and $g(x,y)$ and two integers $a,b \in \mathbb{N}$, we define the 2D discrete convolution as

$$f * g = \sum_{k=-a}^{a} \sum_{l=-b}^{b} g(k,l) f(x-k, y-l)$$

intuitively, we can think of g as a kernel that we "run-through" our function f and for every k,l, the new value we assign to $f * g$ is the sum of all multiplications of points where g lays on top of f.

4.2 Convolutional NN

4.2.1 the outline

A convolution layer is a layer in our architecture that operates convolution instead of just summing up weights (like we saw in the MLP architecture), that is - the values of the $i'th$ layer L_i is given by convolving the previous layer L_{i-1} with some chosen kernel function g, i.e $L_i = L_{i-1} * g$ (up to bias addition and non linearity) . our weights would be the values of the kernel, and these would be (along with the biases) the parameters which we aim to learn in the iterative learning process. Generally speaking, the transition between layers in such architecture would be

- apply N dimensional convolution - same as applying dot product to the input with the kernel
- add bias
- apply some nonlinear activation
- Optional: apply pooling - sample every $j'th$ pixel, for some j (see below)

note that compared to multi layer fully connected NN, applying convolution greatly reduces the parameters of the network, as the matrix G possesses "only" n different values, compared to fully connected layer that applies a matrix with n^2 parameters. Having said that, in convolutional layers it is not clear a-priori what a neuron really are. Most Literature defines the neurons as the rows of G, which are neurons with shared weights (as $g_1, ..., g_n$ are the same for every neuron, with some permutation).

It is common to think of the convolution layers as layers that extract features that are increasingly concrete. For example, the first layers would be responsible for only edge detection, and deeper layers may focus on features that are less vague, such as specific features of a face if we try to detect faces.

4.2.2 pooling layer and strides

a pooling layer reduces the size of the image, which may be relevant for example if our input is too large to be fed through our network efficiently. Say we wish to scale down our input by a factor of 2 - few variations exist

- max pooling - chose a 2×2 section in the input (for example grid of 2×2 pixels) and choose only the pixel that has maximum grey-scale. One downside to this is that we lose the information of which pixel was indeed the one with the max grey-scale.
- average pooling - take the average of your input points in a given window

stride, on the other-hand is the process of sub-sampling our input. this is mostly implemented inside the convolution process - instead of shifting our kernel one value at a time, we shift it two or more values

4.2.3 the architecture

for a more rigorous description, lets consider the case of 2D convolutions applied on images. An image is a tensor that has dimensions $w \times h \times c$, where in most cases c represents `Red,Green,Blue`, therefore it satisfies $c = 3$. A convolution layer receives the image as input, and generates an output that is called an `activation map`, which is a tensor with dimensions $w' \times h' \times c'$, which is the product of convolving with a kernel with dimensions $w_g \times h_g \times c \times c'$. What this actually mean is that we map every cube with dimensions $h_g \times w_g \times c$ to a single pixel in our output, and we perform this c' times, for c' kernels. figure `4.1` illustrates a concrete example

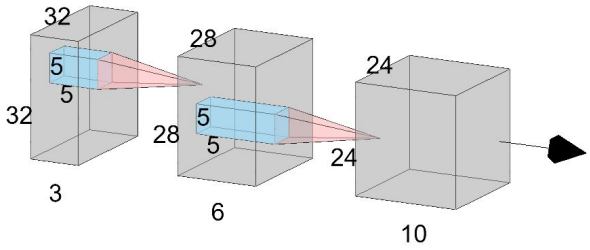


Figure 4.1: left to right - $layer_1$ ($32 \times 32 \times 3$) convolved with $kernel_1$ ($5 \times 5 \times 3 \times 6$) outputs $layer_2$ ($28 \times 28 \times 6$) convolved with $kernel_2$ ($5 \times 5 \times 6 \times 10$) outputs $layer_3$ ($24 \times 24 \times 10$. notice there is no padding, so the width and height decreases, and between each layer we may perform some non linearity σ

note that (when no padding is being made) the dimensions of the $i'th$ tensor are fully determined by the dimensions of the kernel and the previous tensor $i-1$:

$$w_i = w_{i-1} - w_{g,i-1} + 1,\ h_i = h_{i-1} - h_{g,i-1} + 1 \text{ and } c_i = c_{g,i-1}$$

this is a mere result of how convolution works.

4.3 Other Invariances

Setting aside time and space, it is easy to come up with more cases for which our model should be invariant to. For example, two images of a cat - one that is dark and one that is bright, should always be classified the same way. This means that I want my system to be invariant to changes in lighting. A good network might classify such cases properly without apriori intervention, but we can make it's life easier if we apply some smart preprocessing to our data, say normalize each pixel beforehand. As this may indeed work, it is more common to normalize one of the layers and not the images (or add a new layer that simply normalized its input). This process/layer is known as `Local Response Normalization Layer`, and is formally defined as the following mapping:

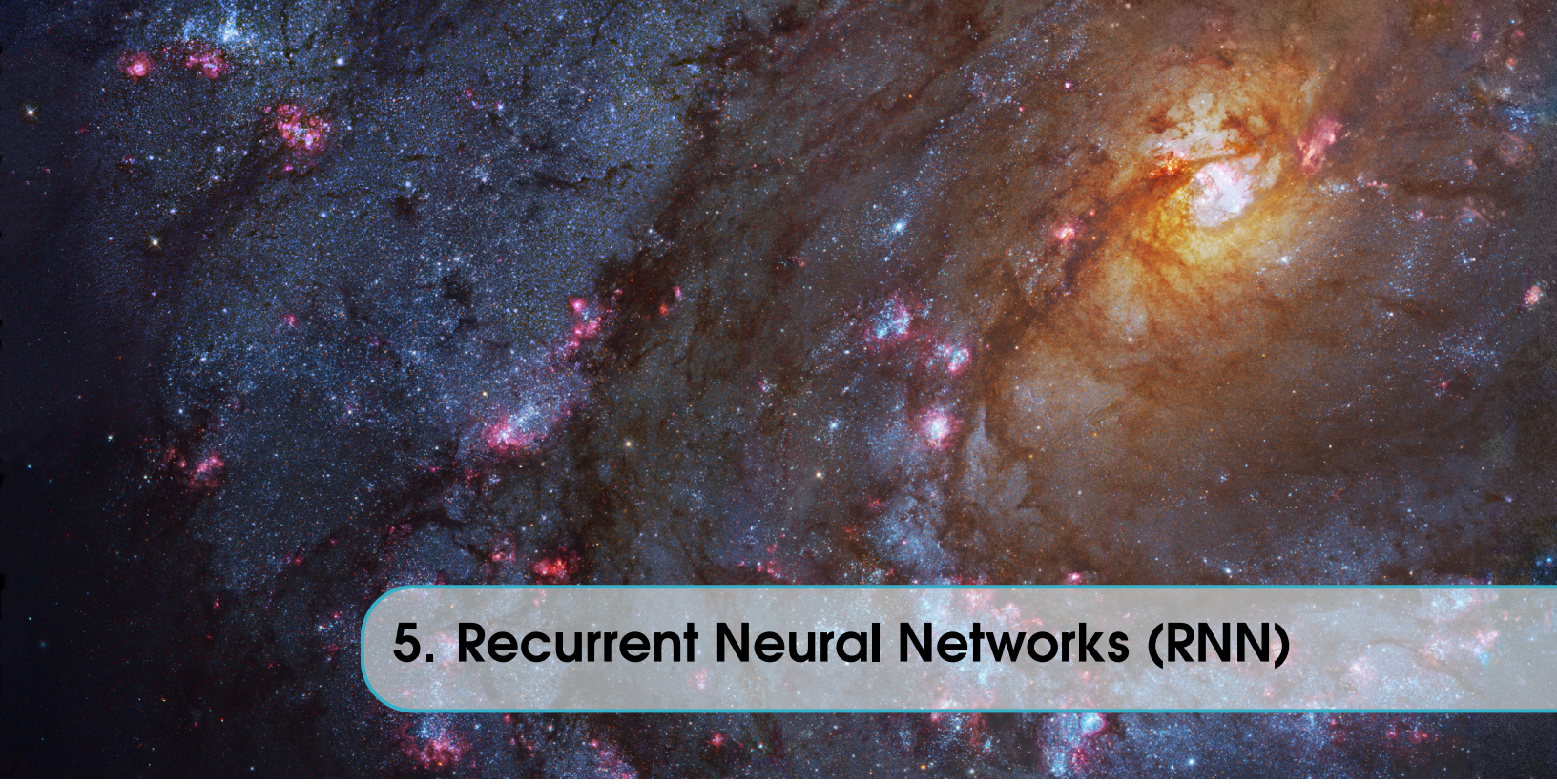
$$R_i(x) \to \frac{R_i(x)}{(c+\alpha\sum_j R_j(x))^\beta}$$

where c, α, β are some predefined parameters, and $R_i(x)$ represents some response of sample i

another example is when we wish that our results would satisfy some relation - say we output (x,y), but the only relevance is that (x,y) would be positioned on a circle defined by $r = \sqrt{x^2+y^2}$. What we can do is map the predicted position (x,y) to $\frac{(x,y)}{\sqrt{x^2+y^2}}$, which formally means (in terms of

invariance) that all dots (x,y) on the same ray should land on the same point where the ray intersects the circle, and for that reason we add the normalization.

Lastly, there's also the case of projecting our data to values bigger than 0 that sums to one, this could be done using the mapping of our output vector $x = (x_1,...,x_n)^T$ to $softmax(x)$



5. Recurrent Neural Networks (RNN)

As of now, we only considered inputs that were fixed and predefined, that had similar dimension to the matrices that made up our architecture (either in a fully connected MLP or in Conv NN). Let us relax that assumption, and discuss new types of inputs.

5.1 Sequence signals

we consider Sequence signal as any signal that has the following properties

- the prediction at some time t depends on history (say, time $t - \tau$)
- the input may have varying length

for example, Google Translate receives its input as a sequence signal, as the translation of some sentence relays on the words that were previously inserted, and the sentence provided could be of any length.

5.2 Recurrent NN

the main idea would be that we can receive an input, perform some algorithm to it and then output it to a future version of our model, that would also be able to receive future input, as well as the prediction of the previous model. If so, for every time stamp t, we

- receive an input x_t
- produce a prediction y_t
- produce a memory hidden state h_t that can be transferred to future states

generally speaking, we can think of this model as some generic predictor h that is constantly receiving inputs and constantly predicts outputs, or (unfolding) as a sequence of predictors, as the below figure describes. This model is known `Elman network` notice that every input x_i and every output o_i are of constant size, but the number of such inputs and outputs is unknown, and isn't necessarily constant.

usually we will look only at the output at the very end, and back to our Google Translate example, this is intuitively clear as a portion of a sentence may not have any meaning up until it is finished.

we can formally write h_t as

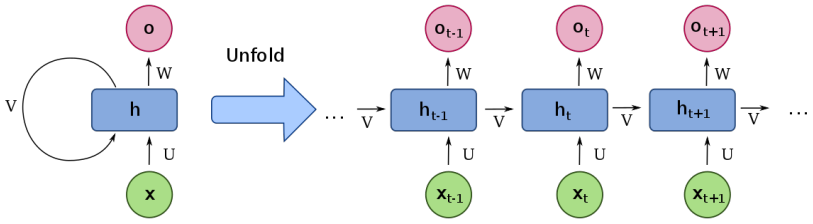


Figure 5.1: Elman network diagram. U, V, W are some matrices that we multiply x_i, o_i and h_i with (aka the network parameters).

$$h_t = \sigma_h(U_h x_t + V_h h_{t-1} + b_h)$$

where W_h and U_h are some matrices that we learn in the iterative learning process (they are not different per t, as we do not know how big t is), and b_h is the bias. our prediction o_t would be

$$o_t = \sigma_y(W_y h_t + b_y)$$

In general notation we could also write $h_t, y_t = N_\theta(h_{t-1}, x_t)$

5.2.1 RNN architectures

One key feature of Elman's network model is that we have the freedom to play with the number of inputs/outputs. In most cases we distinguish between the following designs

- `one-to-one`: the input and the output are of fixes size, this is similar to models we already saw before.
- `one-to-many`: fixed size input, and arbitrary size output. for example we input an image and output its textual representation
- `many-to-one`: exactly the opposite. for example - trying to predict stock price given many future values
- `many-to-many`: for example, translating Hebrew to English.

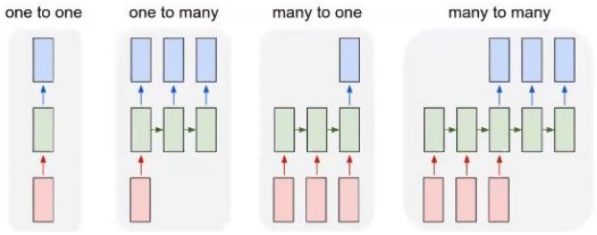


Figure 5.2: Various RNN architectures.

5.3 Back Propagation Through Time (BPTT)

In regular back-propagation, The loss was only a function of the weights and biases, and therefore the derivatives of the loss were with respect to the weights and biases. On the other hand, in RNN

there are more inputs to the loss, so we have to be careful when deriving with respect to the weights. Lets denote the model prediction and hidden state as

$$h_t, o_t = N(\theta, h_{t-1}, x_t)$$

that is, the network outputs both h_t and y_t. we also denote $N_h = h$, $N_o = o$ (as N outputs a tuple). from here, we may choose some lose function L (that could, for example take the squared differences between N and some labels y), but regardless of how we chose L, there would be a tern in which we specifically derive the network's prediction, so let us focus on that term

$$\frac{\partial h_t}{\partial \theta} = \frac{\partial}{\partial \theta} N_h(\theta, h_{t-1}, x_t) = \frac{\partial N_h}{\partial \theta}\frac{d\theta}{d\theta} + \frac{\partial N_h}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial \theta} + \frac{\partial N_h}{\partial x_t}\frac{\partial x_t}{\partial \theta} = \frac{\partial}{\partial \theta} N_h(\theta, h_{t-1}, x_t) + \frac{\partial}{\partial h_{t-1}} N_h(\theta, h_{t-1}, x_t)\frac{\partial h_{t-1}}{\partial \theta}$$

where $\frac{\partial x_t}{\partial \theta} = 0$ because the input is not a function of the network parameters. Now comes the tricky part, as we recursively expand the values of $\frac{\partial h_t}{\partial \theta}$ for every t to achieve the final results

$$\frac{\partial h_t}{\partial \theta} = \frac{\partial}{\partial \theta} N_h(\theta, h_{t-1}, x_t) + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t} \frac{\partial}{\partial h_{j-1}} N_h(\theta, h_{j-1}, x_j)\right)\frac{\partial}{\partial \theta} N_h(\theta, h_{i-1}, x_i)$$

which can be written in shorter notation as

$$a_t = b_t + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t} c_j\right) b_i$$

where

$$a_t = \frac{\partial h_t}{\partial \theta},\ b_t = \frac{\partial}{\partial \theta} N_h(\theta, h_{t-1}, x_t) \text{ and } c_t = \frac{\partial}{\partial h_{j-1}} N_h(\theta, h_{j-1}, x_j)$$

this gives us a way to derive the loss - suppose $L = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t)$, for some loss function l (say, $l = l_2$) and T samples. The gradient is given by

$$\frac{\partial L}{\partial \theta} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial \theta} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial o_t}{\partial h_t}\frac{\partial h_t}{\partial \theta}$$

note that l is some loss function over o_t, so we can easily derive it with respect to o_t. Furthermore, $o_t = \sigma_y(W_y h_t + b_y)$ so we can derive it with respect to h as well. the third factor is given from the calculation above, and this concluded what we wished to achieve.

Though the derivation of $\frac{\partial h_t}{\partial \theta}$ from above can be computed fully, for large T this gets extremely long. There are some strategies for dealing with such problem, but we will not discuss them in the course

5.4 Vanishing/Exploding gradient problem

when we look at the term $\prod_{j=i+1}^{t} c_j$, it is reasonable to think that if c_j is relatively big, the product would converge exponentially. This will obviously cause a problem, as our gradient iteration will not bring us towards the solution (in fact, it will probably won't provide any solution). On the contrary, if the term c_j is sufficiently small, we will face the opposite problem of vanishing gradient, and the network will work hard to converge. the problem described above isn't solely related to RNNs. For example, a very deep NN could encounter vanishing gradients. To minimize the effect, `skip connection` was introduced. This means, in general, that some input x (of some layer) would not be multiplied (or convolved) with the layers that follows him , and instead would be transferred directly deeper down the network blocks.

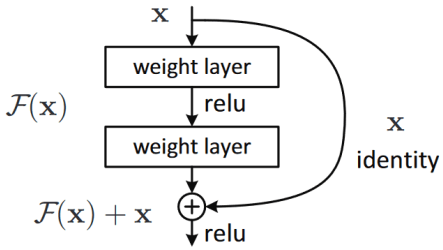


Figure 5.3: skip connection diagram

5.4.1 First solution: Long-Shirt-Term-Memory (LSTM)

LSTM is an architecture that was introduced to try and reproduce the concept of skip connections in RNNs. the model itself is comprised of a concatenation of identical "cells", and each cell has a few components that are in charge of various applications

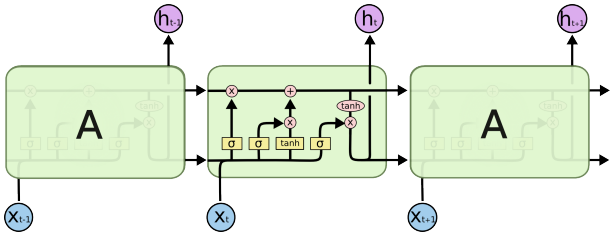
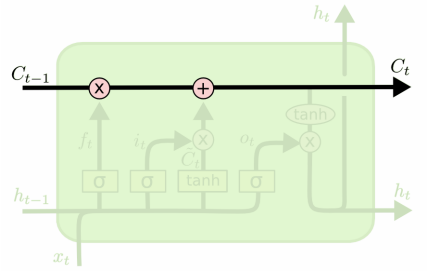


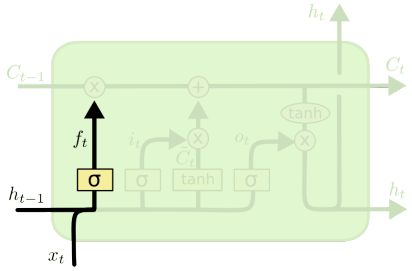
Figure 5.4: LSTM is a concatenation of cells

let us break the cell to its main components:

1. `Cell State`: represented with c_t, is the analog of the `skip connection`. The cell state is to be thought of as the "memory" that it passed along from previous cells. With that intuition in mind, notice that c_t can be multiplied by a number $\in (0,1)$ (output of some `sigmoid`), and this will allow us to control how much from past iterations we wish to "remember" (0 - forget the history and 1 let it pass fully). Finally, our c_t is added with "+" to our processing of h_{t-1} and x_t, which is important as we remember that when deriving with respect to our network parameters θ, $\frac{\partial c_t}{\partial \theta} = 0$, and therefore c_t will not contribute to vanishing gradients.

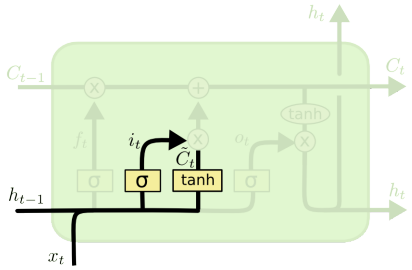


2. `Forget State`: both the current input x_t and the output from the previous cell h_{t-1} are inserted as input to a layer that outputs some value $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \in (0,1)$ ($[\cdot,\cdot]$ means concatenation).This value, as described before, is what chooses whether to "forget" or "remember" c_t.



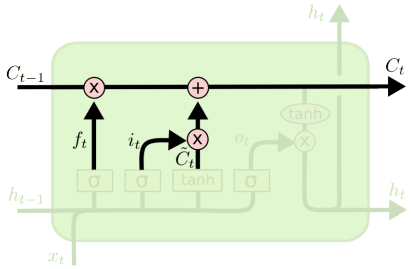
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

3. `Update (Input) State`: the current input and previous output can update the previous memory by adding to it new memory $\tilde{C} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$. Before $\tilde{C}$ is added to the previous c, we also calculate $i = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ which will provide a percentage for how much of $\tilde{C}$ we wish to transfer. We use tanh as it allows us an update that is both addition and subtraction (as $\tanh(x) \in [-1, 1]$)



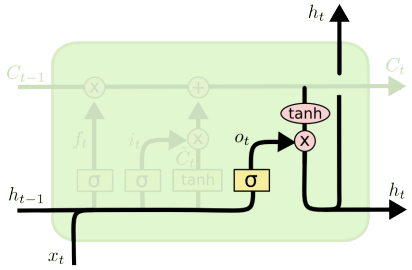
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

4. `Forgetting and Updating Cell State`: This state is responsible of updating the memory c given the outputs of the Update state and Forget state, that is it sets $c_t = f_t \times c_{t-1} + i_t \times \tilde{C}_t$, where $\times$ is `element-wise` multiplication.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

5. `Output State`: The last state is responsible for generating the output of the current cell, o_t and the hidden state h_t. specifically we have $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$ and $h_t = o_t \times \tanh(c_t)$, which indicates that the hidden state is some factor of the output with the previous layers memory.



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

—

Provided an LSTM architecture, the problem of vanishing gradients greatly reduces, as the cell state can pass either a long or short memories without changing them, which is what skip connection is

all about. This allows us to build a deep model with elongated time dependencies. Do notice though that there are still assignments to the parameters of the LSTM that will result in vanishing gradient, and therefore the method isnt bullet proof.
the above could be concisely summarized to the following set of instructions, where notice we've added matrices U, which won't change the architecture (you could either use $W \cdot [h_{t-1}, x_t]$ or $Wh_{t-1} + Ux_t$):

`forget gate` - controls what is kept vs forgotten from previous cell states:

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

`input gate` - controls what parts of the new cell content are written to cell:

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$

`output gate` - controls what parts of cells are output to hidden state:

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$

`new cell content` - is the new content to be written to the cell:

$$\tilde{c}_t = \tanh(W_c h_{t-1} + U_c x_t + b_c)$$

`cell state` - forgets some content from last cell state and writes some new cell content

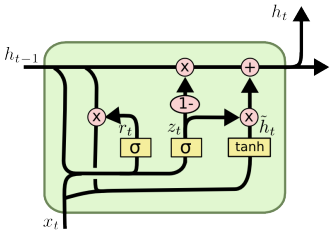
$$c_t = f_t \times c_{t-1} + i_t \times \tilde{c}_t$$

`hidden state` - outputs some content from the cell

$$h_t = o_t \times \tanh(c_t)$$

5.4.2 Second solution: Gated Recurrent Unit (GRU)

The GRU is a recurrent unit that was introduces by `Cho.et al(2014)`, that has similar performance as the basic `LSTM`, though is more compact and has less inner parameters. let us briefly go over its main components:



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 5.5: GRU cell

The first thing to notice is we do not use c_t any more - the hidden state h_t will hold the information that was previously attributed to c_t. More formally, h_t can be either updated (using z_t) or rebooted (using r_t) in the following manner: if we choose to reset, r_t will be set to $(0,0,...,0)$ and the updated hidden state h_t will only be affected by the input of the current time stamp, x_t. eventually the hidden state will be a factor of both the updated state $\tilde{h}_t$ and the previous state h_{t-1}, as a sum with factoring by z_t - if z_t is closer to 1, h_t will be less similar to the previous h_{t-1}, and vice versa (see figure). Also notice that the update is performed using tanh, as to incorporate both addition and subtraction.

6. Attention Layers

As we dive deeper to the realm of deep neural networks, we are facing the realization that getting acquainted with the inductive bias of our network, that is, finding how differently our network tends to learn the data based on its specific architecture, is indeed a key component to the success of the network. A great example to that is the usage of CNNs instead of MLPs, as we had some prior knowledge about our data - we assumed in was LTI, and this allowed us to specifically build our model to use that prior knowledge. We can take this general concept to the next level, as in broader terms, we would also like our network to distinguish what is essential and what is bland, that is, equip our network with some way to divide its "attention" between samples that have significant meaning, and samples that does not.
In this chapter we will see how the intuition above was translated to what is called an `Attention Layer`

6.1 The problem with encoder-decoder models

encoder-decoder are a general term to models that have (no surprise here) two phases

- `encoding` - we receive an input (say, some sentence), and encode it to hidden states. This phase is finished when the input was fully processed (for example, when we have reached the end our the sentence)
- `decoding` - we decode the data possessed in the hidden layers to some meaningful output (for example, some translation of text)

The problem is that our input may very well be extremely long, with some complex dependencies inside of it, but still, at some point t in time we expect our encoder to give us some output vector o_t of constant length that should contain all that information. The problem here is evident, and indeed it was shown that large and complex inputs weren't successfully decoded to a meaning-full output (even when using GRU)
another example of the complexity of the problem arises when discussing the following learning task: given some image input, provide a textual representation to it. Why is this so difficult? consider an image of a girl throwing Frisbee. Even if the image is very clear, it may consist of various other

parts - for example, there ought to be some background (trees maybe?), etc.. Long story short, as we feed our input through some fancy deep vision CNN concatenated with an LSTM cell, we still output some vector that should in theory represent the entire image (and not only the girl throwing Frisbee), What we would have wanted is some way to tell the network which parts of the image are more relevant than others, aka, to which pixels should the network pay more attention

6.2 The solution - Attention Layer

Attention Layer is essentially a mapping/table, that given keys $\{k_i\}$, values $\{v_i\}$ and a query q

- compute the proximity d_i between q and every key k_i
- apply `softmax` to define a probability α_i that signifies the relevance of each value
- produce some soft prediction, mean value with respect to the distribution

several variants exists, and the simplest one is

$$d_i = \frac{\langle k_i, q \rangle}{\sqrt{N}},\ \alpha_i = softmax(d_i) \text{ and } v_{out} = \sum_i \alpha_i v_i$$

where N is the size of the table
or shortly in matrix form, where q is a query vector, K is a matrix where each column is a vector of keys and V is a matrix where each column is a set of values:

$$v_{out} = V^T softmax(Kq)/\sqrt{N}$$

another option is to concatenate k_i and q, and add some learnable weights (a vector w, and a matrix W). In that case $d_i = w^T \tanh(W[k_i, q])$, where we use tanh to get both positive and negative values.
lastly, there's also the variant of

$$d_i = \frac{\langle W_K k_i, W_Q q \rangle}{\sqrt{N}},\ \alpha_i = softmax(d_i) \text{ and } v_{out} = \sum_i \alpha_i W_V v_i$$

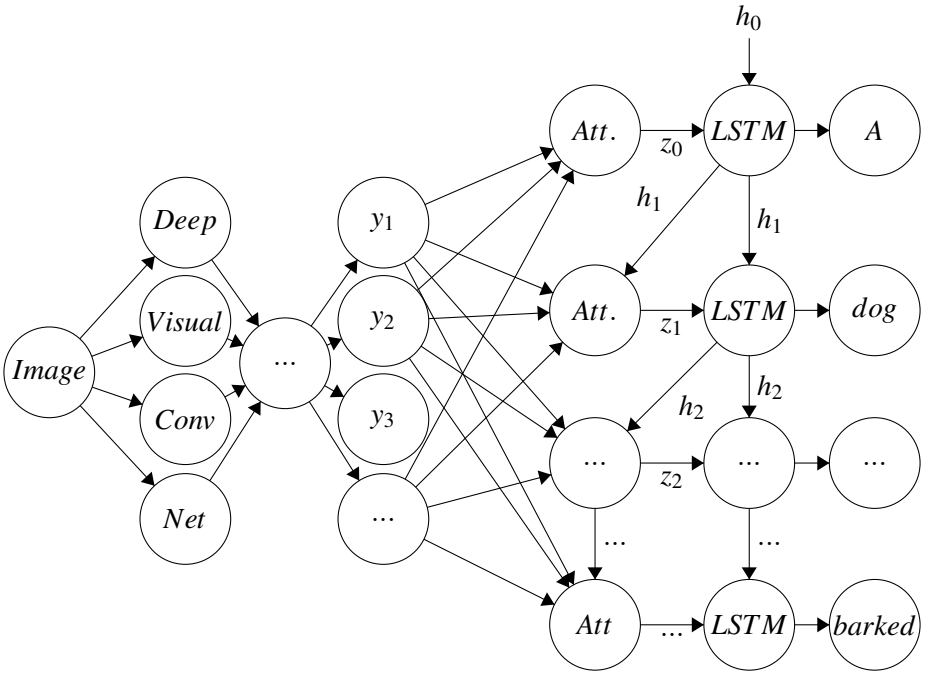
which is a common variant, that allows k and q to have different dimensions.
lets see how this layer can be added to the task of image captioning to provide significantly improved results:

6.2.1 Image Captioning

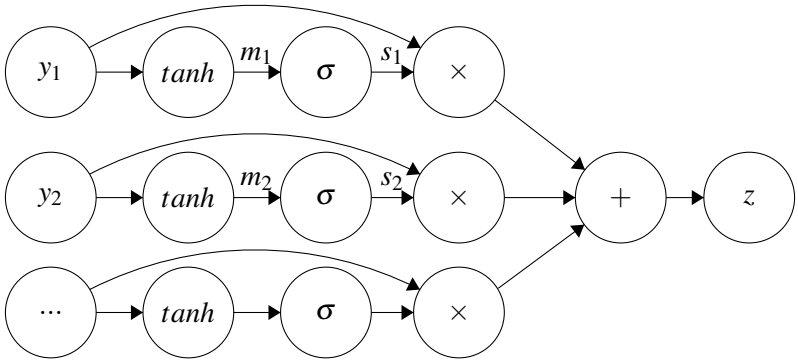
based on the paper `Show, Attend and Tell`
As a reminder, the task in hand is to provide meaningful caption to a given image. We faced the problem of encoder-decoder, in which some of the problems were that the encoding is being made on the entire image, without taking into consideration different regions that may be more relevant. In this approach, we will define attention layers that will do just that. Consider the following architecture (See figure)

- the input is fed to a Deep Vision CNN
- we zoom in on some (non-output) layer that outputs a vector y
- every element y_i is to be considered both as key and as value, and is connected to Attention models that outputs some z_j
- every z_j is fed as sequential input to an LSTM cell
- the LSTM cell outputs a hidden state h_i and some output (for example, a word), where h_i is inserted back to the Attention layer as a query q ,

this can be visualized as



where the Attention layer is as followed ($\sigma \equiv softmax$):



Where in fact, the input to tanh is both y_i and the output of the $LSTM$ cells, which is the hidden state (omitted from the diagram to avoid cluttering). Notice that every attention layer is provided with all $\{y_i\}$. under these notations, we have $m_i = \tanh(y_i W_{y_i} + cW_c)$, where c is really the hidden states memory h. we applied tanh so that very high/small values would have small differences. This is not obligatory, and we could also use inner product, as we have described in the previous section. these m's are then fed to a softmax function $s_i = e^{m_i} / \sum e^{m_i}$, and finally, the output z is the expectation of y under the probabilities provided as s_i (was denoted α_i before), i.e $z = \sum s_i y_i$. It is important to understand that the output of the CNN is provided without the notion of a future sentence that is to be constructed - it only gives us the information for the various key features of an image. the attention layer takes that data and combined with the LSTM cell provides a mapping between the features in the image and their appropriate positioning in a sentence.

R taking $v_{out} = \sum \alpha_i v_i$ is known as `soft` Attention. on the other hand, `hard` Attention samples one of the v_i according to α_i. This may be relevant when taking a single value makes more sense than averaging

■ **Example 6.1 — Neural Machine Translation (NMT).** The general topic of translation from one language to another is known as Neural Machine Translation (NMT). as this was previously discussed, let us tackle the task of translating sentence in which the words may be related to previous or succeeding words. with RNNs, one approach would be to parse our input sentence both from beginning to end and vice versa, and for every time step generate a feature vector. From here, all feature vector would be fed to an attention mechanism, where it will help with deciding which words are the most relevant based on the words that have already been read.
such mechanism generates different weights to the attention layers in every time stamp, and a nice representation to this would be to look at a plot in which the vertical axis represents the translated sentence, the horizontal axis represents the given sentence, and the color is associated with every pixel represents how the weights distribute among the input words. consider a trivial translation

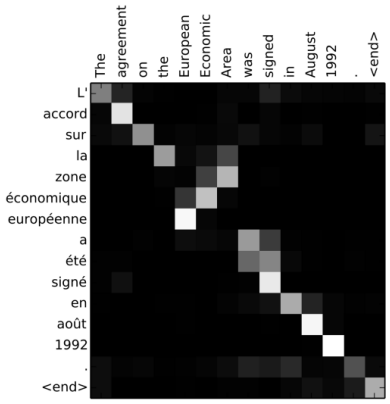


Figure 6.1: Alignment matrix - from English to French

mechanism, for which every word in the input is directly translated to a word in the output. the alignment matrix would be strictly diagonal, as there is one to one correspondence between every input word to a single output word. figure 6.1 demonstrates that using attention layers this is not the case - and some words were translated with respect to two or more words, so the matrix isn't diagonal. ■

6.2.2 Multi-Headed Attention

As a word may be associated to various other words in the sentence (for example, in "I gave my dog Charlie some food", the word "gave" relates to "my dog", "Charlie" and "some food"), the association might be different, and there is no apparent reason for why we should some their affect in a single linear combination (as we would using a single attention layer). To do this we would use apply Attention on the same sentence a few times

$$MultiHead(Q,K,V) = Concat(head_1,...,head_n)W_O$$
$$head_i = Attention(QW_{i,Q}, KW_{i,k}, VW_{i,V})$$

It is important to notice that each head is independent and has different learnable parameters that are (in most cases) randomly initialized which also indicates that they will contain different values throughout the learning process

6.2.3 Self Attention

What is common among the following sentences?

- She loves eating dates
- She took him on a date
- What is your date of birth
- Not to date myself, but...

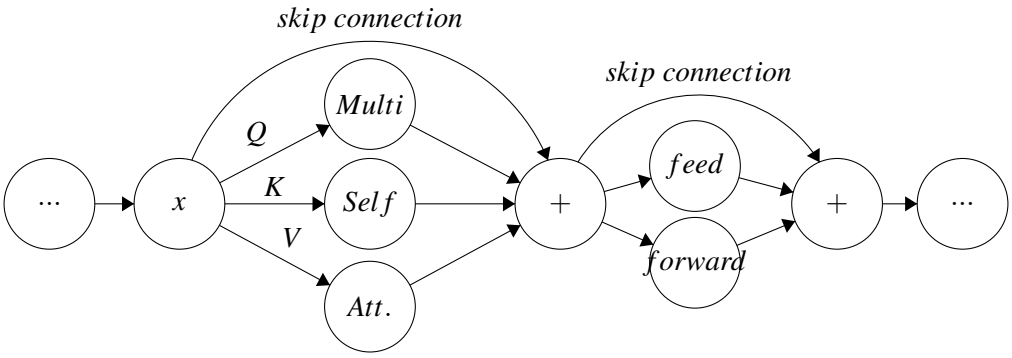
the same word is being used, but its meaning is completely different. Thus, if we wish to translate the sentence, we should relate to the word `date` as if it is a completely new word every time. In other words, to properly translate the sentences above, it would be best if we first understand its contextualization. Self Attention does just that - for a given input x, we define the all the inputs (the query, the key and the value) as x, and output $y = Attention(xW_{query}, xW_{key}, xW_{value})$, where $Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$. The output of the self attention layer reflects how each words related to every other word in the sentence.

6.3 Transformer Network

In our path of describing more and more suitable network architectures, we keep in mind that

- if out input is of varying length: MLP (and CNN) will fail
- if we deal with input that has inner relations with itself (for example, the first word of a sentence is related to the last): CNN would probably fail as well
- if there's no progression axis, that is, only in some time in the future enough input would be given to extract a legitimate and understandable output: RNNs might fail

To provide a solution to all the cases described above, Transformer networks were introduced. Those are MLP networks with the addition of a Multi-head self-attention layer. more specifically, a transformer networks is comprised of blocks, where each block can be represented as followed



that is, we concatenate blocks where each block in itself contains a Multi-head Self-Attention and a feed-forward fully connected network, where the output of the first is fed as an input to the latter (with skip connections as well that should prevent vanishing gradients). the diagram should be read as followed - given some input x (say, a sentence), the multi-head self attention layer generates an embedding for each word, and the feed forward network works on each word embedding independently (This allows us to work on inputs of varying length).
The provided architecture can handle inputs of varying length, and it considers every connection between every segment in out input. furthermore, changing the order representation of our input has no effect (it is invariant), as the self attention layer handles that. Lastly, as the fully connected feed

forward layer parses every segment separately, and the heads in the attention layer are independent, we are able to use parallel computing to speed up the training process.

R as a side note, a performance comparison for various layers is provided
complexity per layer:

- self attention layer: runs in $O(n^2 d)$ as the attention mechanism compares every word with every word, where n is the number of words in the input and d the dimension of every word
- RNNs: $O(nd^2)$, as we parse every word and whenever we get a feature vector for it, matrix multiplication is performed
- Conv NN: for n pixels, d channels and k kernel size, we have $O(kd^2 n)$, as for every pixel we perform convolution which takes the sum of every pixel in the kernel that is both in the image and in the kernel, that is for every channel (d for the kernel and d for the image, though not necessarily equal)

sequential operations

- self attention layer: $O(1)$, as it is automatically aligning every word pair
- RNN layer: $O(n)$ because in every step we perform a calculation on a single sequential input
- CNN layer: $O(1)$ as all the computation is done using single step of convolution

max path length (after how many steps every would would relate to every word)

- self attention layer: $O(1)$ because we match every pair from the beginning
- RNN layer: $O(n)$ because the first and last words demands n steps
- CNN layer: $O(log_k(n))$ as the two most distant pixels will be narrowed down under the same kernel frame only after the image was convolved with the kernel $log_k(n)$ times

6.3.1 Encoder-Decoder Transformer for NMT

As an example to the encoder-decoder architecture, a transformer is presented in figure 6.2. the block to the left is one segment (the encoder), which in itself is a transformer that is very deep (n such blocks). the right segment is somewhat different, though is still considered a transformer (that uses two attention mechanisms). The encoder receives the inputs, and its output is some coding of the input sentence (more specifically, the input was encoded using n Transformers). Following the encoder comes the decoder - the coded data which was the output of the encoder is inserted both as keys and as values to a multi head attention layer within the decoder. regardless of that input, every word in the output of the decoder itself is fed right back to the decoder, and is set as query input to the multi head attention layer. This allowed every word in the output to relate in some manner to the entire sentence (and in every such relation, new information may be generated). The masked multi head attention is an attention mechanism that only matches words with words that was already provided in the sentence (only words that came prior to the currently processed word). the above could be summarized to the following steps

1. the input sentence is inserted to the encoder (all words simultaneously)
2. the input is embedded to some vector representation (using GloVe/Word2Vec for example)
3. the input if fed forward through n Transformers (which are what the encoder is comprised of)
4. the output of the encoder is fed both as keys and values to a multi head attention in the decoder
5. in an iterative process, the decoder synthesises the output sentence as followed
   (a) every single word that was the output of the decoder if first embedded and encoded (in the first iteration, a randomly generated word is initialized)
   (b) we perform masked multi head attention on the word with all the words that preceded it

(c) the output of the previous layer is inserted as query to a multi head attention layer (with keys and values provided from the decoder as previously described)
(d) the output is both: multiplied with a matrix that has dimensions which is the size of the output language (a very big number) and normalized with softmax to achieve output probability AND returns as input back to the encoder, to parse the following words

the loss is defined per word in the target language, and is mostly cross entropy

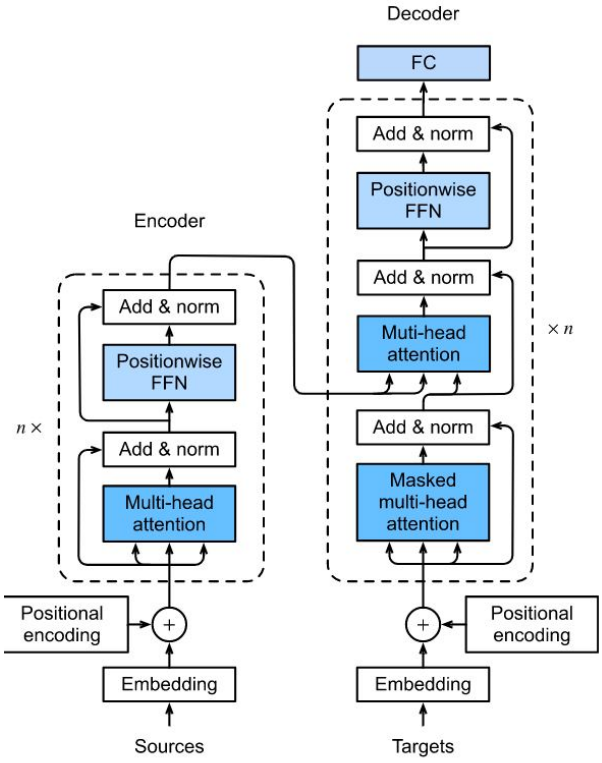


Figure 6.2: Encoder-Decoder Transformer, where the point-wise FNN is defined as $FNN(x) = relu(xW_1 + b_1)W_2 + b_2$ and the positional encoding (which tries to map between every word in the input to some position) was defined as a mapping from the position of a word pos (with corresponding index $2i$ of $2i+1$) to sine and cosine functions - $PE(pos, 2i) = \sin(\frac{pos}{(10^5)^{2i/d_{model}}})$ and $PE(pos, 2i+1) = \cos(\frac{pos}{(10^5)^{2i/d_{model}}})$

6.4 Bidirectional Encoding Representation Transformer (BERT)

The above results brought to life the notion that one can freely concatenate a transformer to provide general purpose network for which the order of the input doesn't matter (it can though, if we add positional encoding), all the elements interact with each other, and the input itself can be of varying length. Lets take a glimpse to the world of BERT, providing a high level description of a general purpose language model:
we use a pre-trained BERT, trained both on next sentence prediction (NSP) and on filling missing words (masked language model - MLM). as for the latter, the notion is to filter out some percentage

of the words in the sentence (i.e replacing say 15% of the words with some arbitrary token) and let the BIRD model fill in the masked words. with respect to the missing words, a loss function is defined. The NSP is being performed while the words are still masked (from the MLM process), and its general process is to choose between two sentence options which is more suitable to positioned before the other. The pre trained models were trained mostly in an unsupervised framework, as most of its data wasn't labeled (though at least for the masking task, no labeling is needed as one could simply compare the masked version with the input). Now, with the trained model in hand, one could, for example, use it as a method of embedding the input (instead of using GloVe or word2vec) in a fashion that also provides context (the word date would be embedded differently with respect to the sentence in which it was given). Another option is to train BERT on our own data, as a some what continuation to the training that was already been done, and provided the already chosen weights. This would provide a great starting point, and would tune the BERT model to suit better to our specific task.

7. Auto-Encoders